Automated use case analysis

Master Thesis

Christian Holmboe and Jörgen Hartman

Department of Computing Science
Umeå University, SE-901 87 Umeå, Sweden

Abstract

Use cases are widely used to capture requirements for software systems. This master thesis presents common use case concepts and explores a method for finding domain object model information in use cases by automated grammatical analysis. A tool for use case analysis that uses our method to find classes, class attributes and class methods is presented. To show a better view of the result, we integrated our tool in Fujaba to emphasize extracted information and show the result in UML diagrams.

Acknowledgments

The authors would like to take the opportunity to thank the following people for support and encouragement during the work with this thesis: Jürgen Börstler (supervisor), Ewa Bergqvist, Hans Davidsson, and Lenita Olofsson (proof readers).

Contents

1 Background **3**
1.1 Introduction . . . 3
1.2 Paper overview . . . 4
1.3 Previous work . . . 5

2 Use cases **7**
2.1 The purpose with use cases . . . 8
2.2 Common use case techniques . . . 9
2.3 Use case formats . . . 13
2.4 Problems with use cases . . . 14
2.5 Finding objects by grammatical inspection . . . 15

3 UseCaseAnalyzer **17**
3.1 Design . . . 18
3.2 LinkGrammar . . . 22
3.3 WordNet . . . 22
3.4 XProlog . . . 24
3.5 Rule creation . . . 26
3.6 Usage . . . 27
3.7 Limitations . . . 30

4 Fujaba **33**

5 Summary **37**
5.1 Discussion . . . 37
5.2 Future work . . . 38
5.3 Conclusion . . . 39

A The Library use cases **43**

List of Figures

2.1 Example of an informal system specification ([EP98]) 9
2.2 Example use cases in UML notation 10
2.3 A use case that describes lending an item in a library. 12

3.1 UseCaseAnalyzer 18
3.2 Analyzer work flow 19
3.3 Example on what the traceability references may look like . . 21
3.4 Words with typed links 22
3.5 A valid sentence linkage 23
3.6 Prolog facts from the sentence: 'The Librarian adds a title to the system.' 25
3.7 A Prolog attribute rule. 26
3.8 A Prolog method rule. 27
3.9 UseCaseAnalyzer main GUI 29
3.10 Dialog for enter or edit use cases 30
3.11 Output from a valid use case 31

4.1 The Analyzer integrated in the FUJABA environment 34
4.2 How to create a new class diagram from a use case 35
4.3 The automatically created class diagram from an example use case 36

Chapter 1

Background

1.1 Introduction

This thesis presents an approach to extracting domain model information from use cases. Use cases are commonly used in software projects to aid communication between customers, requirement specialists and programmers during requirement gathering. A use case is typically a textual description of a scenario that the software should be able to handle. "Withdraw cash" for an ATM banking system or "Register a new book" in a library system are examples of such use cases. The latter use case would consist of a step by step description of the actions that are needed in order to register a new book in the library. To describe the functionality and behavior of a complete system it is necessary to produce use cases describing every scenario that the system will be required to handle. Such a set of use cases indirectly describes the domain model of the system. The domain model describes a subset of characteristics of the problem domain in which the intended system will operate. The specific characteristics that we are interested in for the purpose of this thesis are the *objects*; their *attributes* and *methods*.

The language used in use cases may be any natural language, for example english. Natural languages are effective tools for humans to communicate information about the world. They are effective because they have bindings between the linguistic constructs and the structures of the world that they describe. The linguistic constructs are in turn defined by the language grammar. In our approach to analyze use cases we try to find objects, attributes and methods in the sentences by running rules that match certain linguistic constructs. The rules are based on the language grammar and maps linguistic constructs or combinations of linguistic constructs to objects, attributes and methods. Simplified, one could have rules that map nouns to objects,

adjectives to attributes, and verbs to methods.

Being able to extract this kind of information could be a great aid for developers involved in object oriented analysis (OOA) in use case driven software projects. A tool that reliably can produce a domain model will not be available anytime soon. A tool that can provide a good starting point for OO analysts is however feasible. This thesis explores a possible way of extracting domain model information from use cases by implementing a use case analyzer and integrating it in FUJABA, a feature rich development tool written in Java (see Chapter 4).

1.2 Paper overview

This thesis is structured to present the important concepts of use cases and use case analysis. Furthermore an attempt of utilizing these concepts in a more practical manner is presented.

A brief description of each chapter follows:

- Chapter 1: Gives an introduction to the thesis, why it is of interest and what work has previously been done in this area.

- Chapter 2: Introduces the concept of use cases; what they are, why they are used and what issues they address. This chapter includes information about different views on use cases and problems that can arise when a use case model is utilized as a tool in object oriented software engineering. At the end of this chapter a technique for overlapping the bridge between the use cases and the object model is introduced.

- Chapter 3: Presents the USECASEANALYZER which is a tool for automating *grammatical inspection* between the use cases and the actual object model. This chapter also explains how this is done and introduces the different tools that are used in this implementation. Finally this chapter offers a description of how to use this tool, and the chapter ends with a summary of the limitations of this implementation.

- Chapter 4: Gives a brief introduction to the FUJABA project and presents the USECASEANALYZER as an integrated part of FUJABA. With the help of FUJABA the object model produced by the USECASEANALYZER can be viewed as UML diagrams and further developed in the FUJABA environment.

- Chapter 5: Summarizes the thesis by presenting thoughts about how this tool can be refined and expanded in the future to produce better results. Finally in this chapter follows a discussion about what conclusions where made in the process and what advantages and disadvantages this type of tool has.

1.3 Previous work

This thesis is a continuation of a previous thesis by Fransson and Larsson ([FL02]). Fransson and Larsson used LinkGrammar (see Section 3.2) and WordNet (see Section 3.3) in combination with a regular expression based rule scheme to identify objects, attributes and methods in use case text. The work done by Fransson and Larsson has proven to be a good foundation and a source of inspiration for this thesis, even if the design of the UseCaseAnalyzer presented in Chapter 3.1 is fundamentally different from its predecessor.

Fransson and Larsson use a mixed client-server design model. In order to parse a sentence the client first send the sentence to a server. The server is used to interface with LinkGrammar and WordNet and sentences sent to it is passed through these tools and the results are sent back to the client. The client applies two rule sets to these results, one for *Attributes* and one for *Methods*. The rule sets determine what parts of the sentence is to be interpreted as *objects*, *attributes* and *methods*. The server is implemented in C++ and requires a Unix or Linux environment in order to compile and run. The client is implemented in Java and is tightly coupled with its graphical user interface (used for use case insertion and manipulation). These are major problems if the system was to be integrated in Fujaba, since the integrated system must run on non-Unix platforms and the integration should rely on Fujaba's user interface rather than its native interface. Another problem noted by Fransson and Larsson is that their implementation does not take use case actors into account. Actors should generally not become objects in the system since they represent entities interacting with the system rather than parts of the system itself. These problems are addressed and solved by the UseCaseAnalyzer.

A similar approach with LinkGrammar, WordNet and regular expressions is offered by Åberg and Svensson ([ÅS]) to parse descriptions of traffic accidents as a part of their traffic accident visualization tool CarSim. The CarSim projects ambition is to aid insurance companies facing the task of processing car accident insurance claims by analyzing the accident report and visualizing the accident in a computer simulation.

Böttger et. al. ([Böttger+01]) use a tool implemented in Prolog called RECOCASE to analyze use cases. RECOCASE makes use of LINKGRAMMAR and lattice theory and requires for use cases to be expressed in a strictly controlled language.

Juristo, Moreno, and Lopez has proposed an approach to extract object model information during OOA from system specifications written in a controlled language ([JML00]).

D. Rosenberg also presents a method of extracting domain model information from use cases by grammatical inspection ([Rosenberg99]).

Chapter 2

Use cases

The notion of *use cases* and *use case models* was introduced in 1992 by Ivar Jacobson ([Jacobson92]). At that time it was common to specify requirements by listing them in a monolithic document in a more or less structured way. The requirement document could grow to span several hundreds of pages even for moderate systems. The combination of large volume and lack of natural organization made the requirement document cumbersome to work with. Problems often arise in the form of conflicting requirements, difficulties for developers to grasp the requirements, and difficulty to change the requirements once they were set. Jacobson and his team at Ericsson tackled this problem and came up with a method for capturing requirements that was fundamentally different from the traditional requirement listings.

Instead of specifying the requirements for a future system as a flat list of the required functionality, a use case model defines the functionality of the system by describing how the different parts of the system work together. The parts are called use cases and describe a piece of functionality in the system. Each use case describes how its functionality is carried out, step by step, with details of who initiates the use case and what the result of the use case will be. The step by step description describes a complete and meaningful flow of events through the system. In use case terminology the entity that initiates a use case is called an actor. An actor is often a human user but may also be an external system or some other initiator.

The use case model is only concerned with the functionality of the system, not the design, i.e its aim is to answer the *WHAT* and not the *HOW* questions. This distinction is important since mixing functionality information and design assumptions will lock the functionality to the assumed design details, making the use case model a less flexible tool.

Use case driven development has now taken a secure position in software engineering and is becoming an increasingly popular way of developing

software systems.

The remainder of this chapter will discuss the purpose with use cases and some common use case techniques and formats. It will also present some of the most common problems that can arise when using use cases in software development. Finally in this chapter a special technique to find domain objects in use cases called 'grammatical inspection' is described.

2.1 The purpose with use cases

Use case driven development addresses many of the problems found in traditional requirement engineering and documentation techniques. Most of these problems stem from the difficulty to communicate the essence and ideas behind something as complicated as a software system. The main idea behind use case driven development is to let the use case model be an efficient communication tool. A use case model capture and relay the meaning and function of a system in a clear and consistent way between its stakeholders, users and developers. It is paramount for the success of a system that both users and developers understand each other and the system they are building. This lean communication does not happen automatically but the nature of use case driven development makes it easier to accomplish.

To understand why communication between users and developers is difficult, one needs to understand that users and developers look at a software system from different points of view. A user knows what the system is required to do and the developer knows how it could be done. This holds true in the ideal case. In practice the user often has a general idea of what the system should do, but many details may be unknown or simply not considered yet. Developers often try to anticipate the needs of the user and might develop aspects of the system that were not asked for and therefore may never be used. To add to the complexity there are several users and developers with their own individual sets of, some time conflicting, expectations and ideas regarding the system.

A use case model provides common ground for users and developers where aspects of the software system are defined in a way that both parties can understand. The users are forced to be precise by describing each domain process that the system is required to handle, step by step in use cases. This step by step description forms a clear requirement framework that tells the developers exactly what the system should be able to handle.

One problem with traditional requirement listings is that they do not encourage a distinction between functionalities and design assumptions, and rely on the author of the requirements to maintain this distinction. Use

cases do encourage this by explicitly moving focus from design issues to the functionality of the system, thus making it easier to keep the function versus design distinction.

2.2 Common use case techniques

As with most popular process tools the use case model paradigm exists in many different variations. This section describes the practices used in most use case driven development projects. Project specific variations are however common.

Before the analysis phase of a software development project starts, a system specification document is produced by a representative for the end users. The system specification describes the system from the users point of view and defines what it should be capable of handling. As an example, part of a library system specification is shown in Figure 2.1. This document is rarely complete, in the sense that it does not contain enough information for development to start, but captures the main functionality and expectations envisioned by the users. An iterative approach is recommended ([KG00]) to increase the detail of the use cases as the requirements analysis continues.

- It is a support system for a library.
- A library lends books and magazines to borrowers, who are registered in the system, as are the books and magazines.
- A library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in poor condition.
- The librarian is an employee of the library who interacts with the customers (borrowers) and whose work is supported by the system.
- ...

Figure 2.1: Example of an informal system specification ([EP98])

In the first iteration the system specification is analyzed and candidate use cases and actors are identified and named. This is often done by simply looking at the specification and trying to isolate the different scenarios

and functionalities, sometimes with the help of grammatical inspection. Use cases, actors and their relations are documented on a high level, often by using UML[1] notation. Figure 2.2 shows some of the use cases and actors found in the library system example. This first set of use cases are very general and are sometimes called facade use cases ([KG00]). The reason behind the facade use cases is to provide a sketchy view of the system that can be refined later.

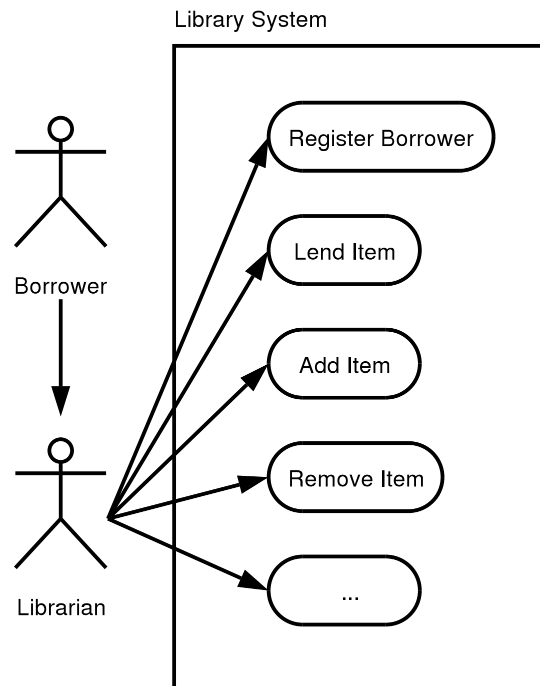

Figure 2.2: Example use cases in UML notation

The facade use cases can contain a brief description of their purpose but may not contain any detailed information. One will often find that this first set of use cases are too general, so they need to be broken down into smaller, more specific use cases. It is a useful to have some informal peer review at this point and to work through the use cases so that they are developed into a solid ground for the next step. When the use cases begin to settle it is time to add some detail.

There are several different use case properties that provide useful information about the system and software projects and organizations often have

[1]UML (Unified Modeling Language) is an standard modeling language not covered in this thesis. More information on UML can be found in [EP98].

their own preferences as to which are used. Some of the most common properties used to detail use cases are:

- Name - A descriptive name of the use case, for example “Register Borrower”.
- Brief description - A short description of the purpose of the use case.
- Precondition - System states that have to be met before the use case can start.
- Postcondition - System states that have changed as a consequence of the use case.
- Actors - External entities and users of the use case.
- Course of events - A detailed description of the steps required to achieve the purpose of the use case.
- Exceptions - Exceptions from the course of events, if any.

Other properties that might be useful to note in a use case is *Version*, *Creation date*, and *Author*. The use case properties are noted for all use cases found so far. Care should be taken to make the use cases as simple to read and understand as possible. Figure 2.3 shows a use case with some basic properties in the library system example.

When all use cases are completed they describe the functionality of the system in great detail while being easy to grasp and understand. It is easy to update the use case model to reflect the system more correctly should any short comings or errors be spotted. The process of producing detailed use cases from a system specification described in this section is a process that often takes place in a changing environment. All persons involved, both developers and users, learn about the system while working with the use cases. New requirements or ideas might emerge from this newly gained knowledge. The process of going from specification to use cases are rarely linear but rather highly dynamic. Different requirements might be on different levels of maturity, some use cases might only exist in a facade version while others are detailed and clear, ready for the design phase which is the next step in the development process (and out the scope of this thesis).

Name: Lend Item
Description: A Librarian registers a loan on the behalf of a borrower.
Actors: Librarian
Precondition: The borrower and item is registered in the system.
Postcondition: A new loan is registered in the system.
Course of events:

1. The Librarian identifies the title and an available item.
2. The Librarian identifies the borrower.
3. The library lends the item to the borrower.
4. The Librarian registers a new loan in the system.

Exceptions:

1. If the borrower has a reservation for the title then the Librarian removes the reservation.

Figure 2.3: A use case that describes lending an item in a library.

2.3 Use case formats

The popularity of use cases and their usage has given rise to a wide variety of use case formats. The differences in the use case formats originate in how the use cases are used in the corresponding methodology. According to Hurlbut ([Hurlbut97]) use case formats can be categorized into three major groups:

- Textual formats
- Graphical formats
- Dynamic formats

The formats categorized in each of these groups have characteristics that suit presentation of different aspects of use cases. Most methodologies make use of formats from more than one of these groups to capture as much information in the use case model as possible.

Textual formats are very common due to the simple yet powerful nature of natural language. Textual representations require little special knowledge to be understood by participants in the requirement gathering phase. Users and stakeholders can therefore learn to work with textual use cases very quickly. The text used to represent the use case is often structured in some way. Some methodologies use an unstructured narrative to describe the use case informally and a structured text for the formal definition. Structuring the text serves to minimize the risk of misunderstandings and ambiguities and is often supported by the use of *use case templates* and *writing guidelines*. The use case shown in Figure 2.3 use a structured textual format.

Graphical formats are used when the presented concepts are more clearly expressed in diagrams than in text. Diagrams usually do a better job of illustrating the 'big picture' than do textual formats. Concepts that are useful to illustrate in a diagram are interaction and structure, for example interactions between actors and use cases and relations between use cases. UML notation is a widely spread graphical format for use cases (among other things) and is a good example of how diagrams can be used to present use case model information. Figure 2.2 shows part of a use case model diagram in UML notation.

Dynamic formats allow the viewer to interact with the use case model, for example navigate through it with hyper links or zoom in or out to get different levels of detail. This kind of use case representation requires software tool support and is not widely used. Zeien ([Zeien95]) proposes a dynamic use case format in the form of hyper linked text where alternative paths and related information is linked together in a web.

There are some alternative use case formats that consist of a mix of the characteristics described above and hence do not fit in any one group, for example *role-playing* with CRC cards or *story boarding*.

If one is to analyze use cases the use case format is an important factor since it dictates what information is available for the analysis. For the purpose of this thesis a minimal textual format has been chosen. The chosen format includes *name of the use case, actors*, and *basic course of events.*

2.4 Problems with use cases

The use case techniques, as good as they may seem, are not without problems and there have been some cautions raised regarding their use in object oriented analysis and design. One of the main issues that a practitioner of a use case technique needs to be aware of is that use cases are functional by nature (i.e. they describe a set of events and functionality). According to Berard ([Berard95]), this is not a problem in itself but needs to be taken with caution within the context of object oriented design. The key point is to remember that objects and functions are not the same thing. If they are interchanged the is a probability that the design will lead to duplication of functionality and result in a design that does not leave much room for code and design reuse. This can lead to that the code have different implementations of the same classes or methods only because several use cases described the same entity with different words. This problem can somewhat be avoided by letting the use case describe only the overlaying view of the system and not use it as a functional view of the system. This view should not lay the foundation for the object oriented architecture but merely be used for further refinement when it comes to finding relevant object entities.

Another similar problem is the violation of information hiding (which is also a cornerstone in OO software engineering) when it comes to writing use cases. In order to write a good use case from the user described functionality, the object (as in system component) and their intended interface needs to be known by the designer but the internal functionality does not (i.e. the analyst should always try to ask *WHAT* the system needs to be able to do and leave out *HOW* the system does it). For example if a use case is describing a library system it needs to know that the system can store loans and books but not how it is stored (e.g. database, files, memory etc.).

When dealing with a large system with a lot of use cases, the use cases are often grouped together into sets of use cases. These sets of use cases are not rarely distributed over different software developers and in turn lay a foundation for even more refined use cases on a deeper level of abstraction.

This often results in a huge amount of use cases where none of the developers are familiar with the whole picture of the intended system, which in turn makes it hard to build a good and accurate object oriented architecture of the system, especially since different people often use their own naming conventions and styles. This also often results in use cases contradicting each other. To resolve this type of problem there has to be a well defined system that can manage all the use cases and a good set of policies, procedures and guidelines concerning the writing of use cases in a large organization.

Another potential problem with the use of use cases modeling is to know when to stop ([Firesmith95]). When designing a complex system, the process often results in a great number of use cases which in turn can produce almost an infinite number of usage scenarios. The big question is how many use cases are needed to describe a complex real world system. Then on top of this, applying object oriented technology makes the simple text book examples and techniques scale badly. Too few use cases results in an inadequate specification, while too many lead to functional decomposition and scattered objects and classes. This potential problem creates a need for software engineers to limit their analysis and designs to cover only the most important scenarios and hope that it will be sufficient to make a complete system.

2.5 Finding objects by grammatical inspection

When the use cases have been created the information gathered needs to be extracted and used in the design of the system. In order to do this a method of some kind is needed. One approach to doing this is to use *grammatical inspection* where the actual text in the use cases is analyzed in a grammatical manner to find domain objects and entities (i.e objects, attributes, methods and different relationships between these entities). The result of a grammatical inspection gives a lot of candidate entities that will not always find a place in the final design and which needs to be analyzed further to decide which ones are going to be real entities and which ones that can be discarded. A basic theoretical approach to make a grammatical inspection can be to treat:

- *Nouns* and *Noun phrases* as objects and attributes
- *Verb* and *Verb phrases* as operations and associations
- *Possessive phrases* as an indication that a noun should be an attribute rather then an object

This theoretical approach results in a list of entity candidates that actual classes, attributes and methods can be derived from and chances are good to find a large majority of the important domain entities this way. The next step is to refine these candidate entities by passing them through a series of manipulations and conversions that place all names in their common case form and delete all duplicate entities. This step will remove anomalies from the candidate entities. For example in a case where both 'book' and 'books' are candidate classes and 'books' is most likely referring to a collection of objects of the 'book' type. When all duplicates are removed from the list of candidates and all plural terms have been changed into their singular or in the case of a verb changed into their common case (e.g running -> run, spoken -> speak, puts -> put). This refined list of candidates now needs to have unnecessary and irrelevant items removed (i.e. because they are redundant, too vague or represent a concept outside the problem domain). For example the use case "The system shall be able to store book titles" should initially give us a class candidate 'system' (since it is a noun) but that candidate will be removed in this phase since it is vague on what it is and most likely refers to the actual computer system that the design is trying to describe.

Many practitioners of grammatical inspection agree that this technique is most useful in getting a quick start on the domain model, and putting too much effort in it (in regard to man hours etc.) will not result in a notably better model. For more information regarding grammatical inspection and a practical example see [Rosenberg99].

Chapter 3

UseCaseAnalyzer

The UseCaseAnalyzer is an experimental system that uses the grammatical inspection technique to find domain objects in use cases. Use cases can be constructed in many different formats (see Section 2.3) and the analyzer needs to be aware of how the format is defined to make a good analysis. In order to make it less complex, a use case format with a minimal set of properties and a textual composition is used in this thesis. The use case format has the following fields or properties:

- *Title:* The title of the use case.
- *Actor:* The name of the active actor in the use case.
- *Sentences:* One or more sentences describing the sequence of events in the use case.

The UseCaseAnalyzer consists of two parts, one *back end* that performs all the conversion and information extraction from the use case sentences, and one *front end* that makes it easy to insert use cases into the system and to view the output created by the analyzer. For the scope of this work the UseCaseAnalyzer only has one language converter implemented (for the Java programming language) but as the system design is created it is fairly easy to make language converters for other object oriented programming languages as well. Figure 3.1 shows an overview of the UseCaseAnalyzer work flow.

The first step in using the UseCaseAnalyzer is feeding all the use cases into the system. These use cases consist of at least one or more sentences which are used in the *Analyzer* to extract potential objects, methods and attributes. It is important to note that the analyzer is only able to hold

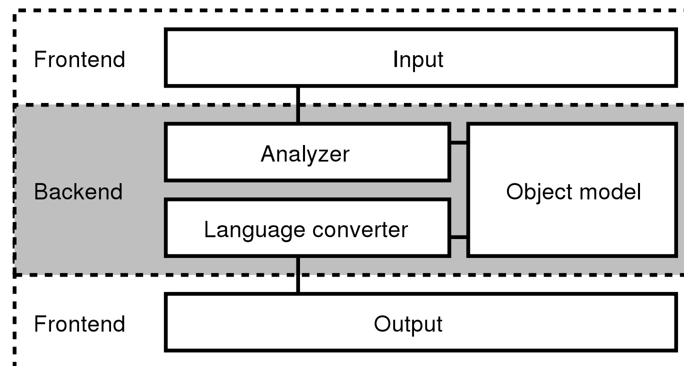

Figure 3.1: UseCaseAnalyzer

knowledge about one sentence at a time and can therefore not extract information that is shared between different sentences. For example a sentence "The cat is green" should give an object candidate 'cat' with an attribute 'green', while the two sentences "The cat is green" and "It has a long tail", would give the object candidate 'cat' with two attributes 'green' and 'tail' (which in turn also is an object candidate with an attribute 'long') but instead extracts to one object candidate 'cat' with the attribute 'green' and another object candidate 'it' with the attribute 'tail.

The *Analyzer* will use the information gathered from all the use cases to create an *Object model* which is the internal representation of classes, attributes and methods (and which also contains all extra information regarding these, e.g. parameters to methods and attribute types). The created *Object model* is then used by an language converter to transform the model to a more human readable form (i.e. Java programming language code). The UseCaseAnalyzer uses the *front end* to show the output to the user of the system. This output can also be used to create UML diagrams in the Fujaba system (see Chapter 4).

3.1 Design

The main goal with the design of the analyzer was to make it as general as possible, so that it wouldn't be tied to any specific programming language. Furthermore it is built in many small and isolated parts that each has it's individual tasks to perform. The purpose of keeping the parts as isolated units was to make a system that is easy to upgrade. This approach makes it easier to exchange or expand one part of the analyzer if, for example, a

better prolog engine has been introduced. This also makes it easy to expand the system to create output for new or other OO programming languages, since all that has to be done is to create a new translator that maps the internal object model to the preferred language.

Figure 3.2 shows the internal parts of the analyzer which corresponds to the *Analyzer* box in Figure 3.1.

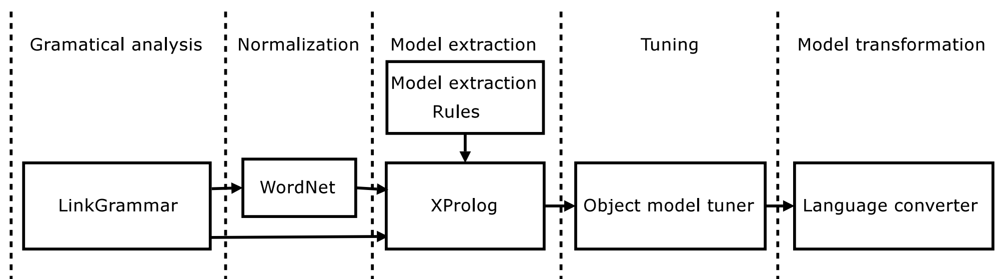

Figure 3.2: Analyzer work flow

All sentences in the use cases are used to create the object model by the analyzer and are fed into the analyzer sentence by sentence. The first thing that happens when a sentence is fed into the analyzer is that it goes through a *grammatical analysis* which aims to classify all words in a grammatic sense (e.g. classify word types; verbs, nouns, adjectives etc.). This is done by the LINKGRAMMAR module which splits the sentences into lists of words with corresponding typed links. For further details about LINKGRAMMAR see Section 3.2. The word list, from the grammatic analysis made by LINKGRAMMAR, is then parsed through the WORDNET module which is used in the analyzer to add information regarding word types on words that the LINKGRAMMAR fails to classify. The list of words then goes through a *normalization* step, which is also done by the WORDNET module, where all words are converted into their common case, for example running is converted into run, cars into car etc. The purpose of this is twofold. The first purpose is to get information on whether a word is in plural form which will, in the case that the entity is an attribute candidate, correspond to an array of attributes instead of a single attribute in the object model. The second purpose is to make all naming of objects, attributes and methods easier, for example to make the sentence 'The dog is running around" into an object candidate 'dog' with a method candidate 'run' instead of the method candidate 'running'. For more information regarding WORDNET see Section 3.3.

The normalized word list, and the corresponding linkage list (from LINK-

GRAMMAR), is then converted in the *model extraction* step into prolog rules and facts and added to the *model extraction rule set* to create the knowledge base of the system. For more information concerning prolog see Section 3.4 on page 24 and concerning *model extraction rules* see Section 3.5. When this is done the model extraction part is ready to start extracting objects, attributes and method candidates that are used in the creation of the object model. Before the actual model is made, each object, attribute and method candidate goes through a *Tuning* step that checks for invalid names and removes unwanted classes, methods or attributes. Like the other parts of this system the *object model tuner* is constructed so that it can easily be exchanged to cover other programming languages or OO model languages as well, but for the scope of this thesis only one was implemented, for the Java programming language.

The final step *Model transformation* is mainly used to translate the object model into other types of model descriptions or an OO programming language. The translation goes through all classes in the extracted object model and for each one of them goes through all attributes and methods and converts them to the desired language.

When the language converter has transformed the object model to the given output type (Java programming language) there can be some trouble to see how the analyzer has managed to get all the different classes, methods and attributes from all the use cases that is used in a system design. The USECASEANALYZER introduces *traceability references* to keep this traceability information by adding a reference to each object, method or attribute that is added to the object model. These references show from which use cases and which sentences a particular object, method or attribute is derived. It is also important if an entity has been derived from different sentences, or different use cases, that this information does not get lost. The USECASEANALYZER handles these references by saving information about the actual use case and sentence that lay ground for a particular entity candidate in the object model. For each case where the analyzer finds an entity (that is already in the object model) a new reference is inserted and saved with this entity. This means that if 'Book' is extracted as an object candidate in two different use cases and within these use cases it makes the analyzer find an object candidate four times, four different traceability references are inserted in the object model.

In the final step of the analyzer, *model transformation*, these references also are converted in a suitable way to follow the output all the way and therefore an *end user* can be able to trace every entity back to its origin. In the Java programming language converter this is handled by inserting one row for each reference above where an entity is declared the first time. Figure

3.3 shows an example of what this may look like in a simple use case.

```
// Class REFERENCE
// Remove Item:The system requests the idenfification bar code of the item.
// Remove Item:The system removes the item, and loans of the item.
// Remove Item:The Librarian identifies the title of the item.
public class Item {
    // ATTRIBUTE - VARIABLES
    // Attribute REFERENCE
    // Remove Item:The system requests the idenfification bar code of the item.
    private Code code;
    // Attribute REFERENCE
    // Remove Item:The Librarian identifies the title of the item.
    private Title title;

    // CONSTRUCTOR
    public Item() {

    }

    // ATTRIBUTE - METHODS
    public Code getCode() {
        return code;
    }
    public void setCode(Code c) {
        code = c;
    }
    public Title getTitle() {
        return title;
    }
    public void setTitle(Title t) {
        title = t;
    }

    // METHODS
    // Method REFERENCE
    // Remove Item:The system removes the item, and loans of the item.
    public void remove() {

    }
}
```

Figure 3.3: Example on what the traceability references may look like

3.2 LinkGrammar

A link grammar is a grammar which main idea is to link words in a sentence together with typed links. Every word has a set of left and right links. In order for two words to connect at least one of the first word's right links must be of the same type as one of the second word's left links. If all words in a sentence are connected without any links crossing each other the sentence is grammatically correct. Note that words do not need to be next to each other to be connected, what matters is that the links match. Grammars for specific languages are defined by specifying link types and associating typed links to the words in the language. Figures 3.4 and 3.5 illustrate these basic link grammar concepts. The UseCaseAnalyzer uses a link grammar software package called LinkGrammar to analyze the grammatical constructs of the use case sentences. LinkGrammar has english grammar predefined so getting linkages for sentences is pretty straight forward. LinkGrammar

also has the ability to make informed guesses about sentences when its predefined grammar is lacking information, for example if an unknown word is encountered LinkGrammar tries to infer the word class of the word from how the word is used in the sentence. Since LinkGrammar is implemented in C, a JNI (Java Native Interface) wrapper library ([ÅS]) must be used to enable it in the UseCaseAnalyzer.

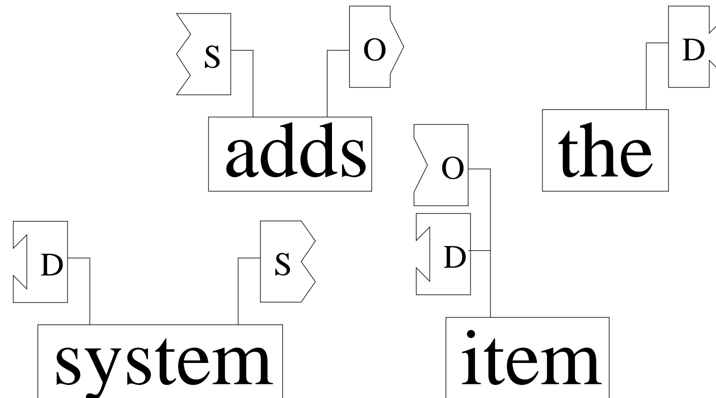

Figure 3.4: Words with typed links

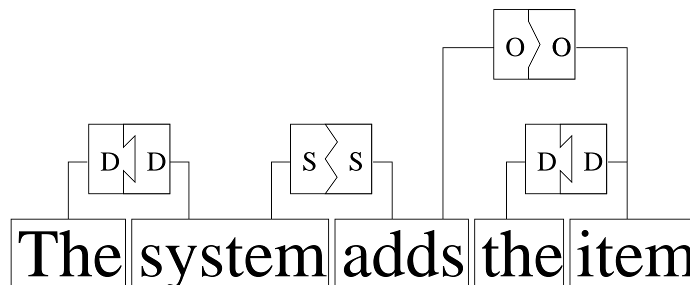

Figure 3.5: A valid sentence linkage

3.3 WordNet

WordNet is a lexical reference system where english nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. These synonym sets are linked by different relations. The basic idea behind WordNet is to have a dictionary that is

searchable conceptually instead of alphabetically. The most common difference between WordNet and a standard dictionary is that WordNet divides the dictionary into five different categories: nouns, verbs, adjectives, adverbs and functional words. This means that if a word is both a verb and a noun it will have its place in both sections in the dictionary. The sections are organized differently, nouns are organized as topical hierarchies, verbs as a variety of entailment relations, and adjectives and adverbs are organized as N-dimensional hyperspaces. For more information regarding WordNet see [Miller+].

The UseCaseAnalyzer uses WordNet mainly to get morphological relation information and to change a word into its common case. For example the verb 'running' will produce a method candidate 'run' and the word 'trees' will produce an array attribute candidate 'tree'. Very rarely LinkGrammar fails to classify the word type of a word, but when it does the UseCaseAnalyzer tries to use WordNet to get the missing information. This is however not totally waterproof since if a word belongs to several categories the computer can not draw the right conclusion on what type the word belongs to and it is left as inconclusive. Fortunately this rarely happens, but when it does it can lead to the analyzer failing to recognize a candidate (object, method or attribute). WordNet contains two other very interesting parts, hyponymy and meronymy. Hyponymy is a semantic relation between word meanings (ISA relation) for example "A car is a vehicle". Meronymy is a *part of whole* (HASA) relation which can for example be "A 'y' has an 'x' (as a part)." or "An 'x' is a part of 'y'." ([Miller+]). These last two relation types are not used in the UseCaseAnalyzer as it is constructed today but in Section 5.2 you can find information on how these two relation types might expand the analyzer further.

3.4 XProlog

Earlier efforts to create a use case analyzer ([FL02]) have used a model extraction rule set where the rules were represented in a static predefined format. This thesis offers a radically different approach to representing model extraction rules. In order to have a more flexible mean of rule representation a first order logic representation of the rules has been considered. Several theorem provers were considered for first order logic inference, but since they were too specialized for mathematic theorem proving or had severe compatibility issues with Java all of them were rejected.

Instead of using a strict first order logic representation Prolog was chosen as representation of the model extraction rules. Using rules that behave

more like a program than a static construct has proven to be very powerful. There are many implementations of Prolog interpreters, the one used in the UseCaseAnalyzer is a package called XProlog by Jean Vaucher. XProlog is implemented in Java so there is no need for additional external installations (as is the case with LinkGrammar and WordNet).

The Prolog program that is used to analyze a sentence is made out of three parts; a base program for rule execution, a collection of model generation rules, and the actual sentence information. The base program consists of support facts necessary to define rules and put forth queries regarding the rules. There are three types of model generation rules:

- classRule(*ClassIdx*) - should return true if *ClassIdx* is part of a potential class object.
- attrRule(*NameIdx*, *Type*, *ClassIdx*) or attrRule(*NameIdx*, *ClassIdx*) - should return true if *NameIdx* is part of a potential attribute to *ClassIdx*.
- methodRule(*NameIdx*, *ClassIdx*) - should return true if *NameIdx* is part of a potential method to *ClassIdx*.

There are also veto rules corresponding to each of the above rule types that can be used to override the creation of classes, attributes, and methods. The sentence information gathered from LinkGrammar and WordNet are converted into five basic types of facts:

- link([*LinkType* | *LinkSubType*], *LIdx*, *RIdx*). - There is a link of type *LinkType*, from word at index *LIdx* to word at index *RIdx*.
- wordIndex(*Word*, *Idx*). - The word *Word* is at position *Idx* in the sentence.
- verb(*Idx*). - The word at index *Idx* is a verb.
- noun(*Idx*). - The word at index *Idx* is a noun.
- adj(*Idx*). - The word at index *Idx* is a adjective.

In addition to the fact types that can be gathered from LinkGrammar and WordNet there is a type of fact that notifies XProlog about actors that are associated with the use case.

- actor(*ActorName*). - *ActorName* is the name of an actor involved with this particular use case.

After a sentence has been converted to Prolog, as shown in Figure 3.6, the base program is able to distill further information from the facts. The link facts contain valuable information about the words they link together, by analyzing their type and subtype many conclusions may be drawn, for example if the nouns are in plural or singular form.

```
wordIndex('LEFT-WALL', 0).
wordIndex('the', 1).
wordIndex('Librarian', 2).
wordIndex('adds', 3).
wordIndex('a', 4).
wordIndex('title', 5).
wordIndex('to', 6).
wordIndex('the', 7).
wordIndex('system', 8).
wordIndex('.', 9).
wordIndex('RIGHT-WALL', 10).
link(['X','p'], 0, 9). % LEFT-WALL -- Xp -- .
link(['W','d'], 0, 2). % LEFT-WALL -- Wd -- Librarian
link(['DG'], 1, 2). % the -- DG -- Librarian
link(['S','s'], 2, 3). % Librarian -- Ss -- adds.v
link(['MV','p'], 3, 6). % adds.v -- MVp -- to
link(['O','s'], 3, 5). % adds.v -- Os -- title.n
link(['D','s'], 4, 5). % a -- Ds -- title.n
link(['J','s'], 6, 8). % to -- Js -- system.n
link(['D','s'], 7, 8). % the -- Ds -- system.n
link(['RW'], 9, 10). % . -- RW -- RIGHT-WALL
noun(5). % 'title'
noun(8). % 'system'
verb(3). % 'adds'
actor(Librarian).
```

Figure 3.6: Prolog facts from the sentence: 'The Librarian adds a title to the system.'

3.5 Rule creation

Creating general rules that define which linguistic constructs are mapped in the domain model requires linguistic skills. Admittedly the authors of this thesis are no linguists so an analytic approach had to be used in order to find suitable rules. A set of example use cases ([EP98]) were chosen as a starting point. The use cases describe a library system in which librarians and loan takers can perform a range of different tasks related to the function of a library. Automated tests were set up to enable validation of the rules to prevent them from conflicting with each other. Link types and their frequency in the use cases were noted. From the frequency information the rules are created empirically by searching for common linkage patterns.

A rule is a Prolog fact and is hence very powerful in its expressiveness. The helper fact *linkChain* is commonly used in the rules to match a chain of links. The Figures 3.7 and 3.8 show examples of actual rules and how they utilize *linkChain* to match a certain linkage.

The rule in Figure 3.7 is quite straight forward. It is an attribute rule that uses *linkChain* to match three words with specific links between them. The links matched are a 'M' link from the first word to the second and a 'J' link from the second word to the third. The underscore after 'M' and 'J' is a wildcard and indicates that any sub type of the links are accepted. A link of type 'M' connects nouns to various kinds of post-nominal modifiers, for example it will connect 'title' and 'of' in the sentence 'The Librarian identifies the title of the item'. The 'J' link connects prepositions to their objects, so it will connect 'of' to 'item' in the same example sentence. From this example sentence the rule will derive an object named 'Item' with an attribute named 'Title'. The type of the attribute can also be set by the attribute rules, but the rule in Figure 3.7 can not make this conclusion from its linkage so the attribute 'Title' will be of the default type (String).

```
%                                     +---Js---+
%                                +--Mp-+        |
%                                |     |        |
% the Librarian identifies.v the title.n of the item.n .
%
attrRule(Name, Class) :- linkChain([['M'|_],['J'|_]], [Name, Of, Class|_]),!.
```

Figure 3.7: A Prolog attribute rule.

The rule depicted in Figure 3.8 is somewhat more complicated but follow the same basic pattern. It is a method rule that instead of matching a single chain of links, matches two different chains that are interconnected. The first link chain matches a 'S' link and a 'O' link. The 'S' link connects subject nouns to finite verbs so it connects 'Librarian' and 'registers' in the sentence 'The Librarian registers the item in the system'. A link of type 'O' connects transitive verbs to direct or indirect objects and will therefor link 'registers' with 'item'. The second link chain consist of a 'MV' link and a 'J' link and starts at the second matched word of the first link chain, that is 'registers' in the example sentence. 'MV' links connects verbs to modifying phrases and, as seen earlier, 'J' links connects prepositions to their objects. The result of the second link chain is a match of the words 'registers', 'in', and 'system'. Together the link chains matches the words 'Librarian', 'registers', 'item', 'in', and 'system'. By the mapping of the matched words and the parameters of the rule it is decided what words will be used for what entity in the object model. In this example the resulting object model will consist

of an object 'System' with a method 'registerItem' that will take an object 'Item' as parameter.

```
%
%                      +--------MVp-------+
%                      +------Os-----+    +----Js---+
%          +----Ss----+              |    |         |
%          |          |              |    |         |
%   the Librarian registers.v the item.n in the system.n .
%
methodRule([Name1,Name2], Name2, Class) :-
                            linkChain([['S'|_], ['O'|_]], [Actor,Name1,Name2]),
                            linkChain([['MV'|_],['J'|_]],[Name1,Mod,Class]),!.
```

Figure 3.8: A Prolog method rule.

As shown in Section 3.4 there are more information such as word types and actors that can be taken into account in the rules. The primary source of sentence information are however the links gathered from LinkGrammar. The rules are defined in a text file and new rules can easily be added and removed by editing the file.

3.6 Usage

The UseCaseAnalyzer comes with a very simple GUI (graphic user interface), mainly because it is the analyzer engine that is important in the scoop of this thesis. The GUI supports the basic functionalities that are needed to test the analyzer engine. Though it can be used directly for creating Java programming language code from different kinds of use cases.

There are basic functions that can be used to create, save, and open projects. A project is a collection of use cases that describes a system, and in each project it is possible to group use cases that are related to each other by creating virtual folders. The UseCaseAnalyzer uses a *tree model* to visualize the project where the root node stands for the project itself, the folders are branches and all the leaves are use cases. The main functionality for the UseCaseAnalyzer lies under the project menu shown in Figure 3.9.

Add Use Case: This alternative creates a new use case and adds it to the tree node that is marked in the project tree in the case that the marked node is the root node or a folder. No use case node can be a parent to another use case node. Nothing will happen if this option is chosen while a use case node is marked in the project tree.

Add Folder: With this option a new folder is created and added to the marked node in the project tree. Only the Project node (root node) or other folders can be parents to new folders. Nothing will happen if this option is chosen while a use case node is marked in the project tree.

Show/Edit Use Case: This action brings forth the use case input dialog (Figure 3.10) for a newly created use case or an old one if a use case is marked in the project tree when this action is chosen. If a folder is marked in the project tree a change folder dialog is shown where it is possible to change the name of the current folder.

Remove Tree node: This action will remove the node that is marked in the project tree. If it is a folder, all its sub folders and connected use cases are removed. This action is irreversible in this version of the system and should be used with care.

Parse node: This action will do an analysis of the marked node in the project tree (and all its sub folders if it is a folder) and create a merged object model for that part of the described system. This is an excellent way to watch what for example one or a group of use cases contribute to the complete object model.

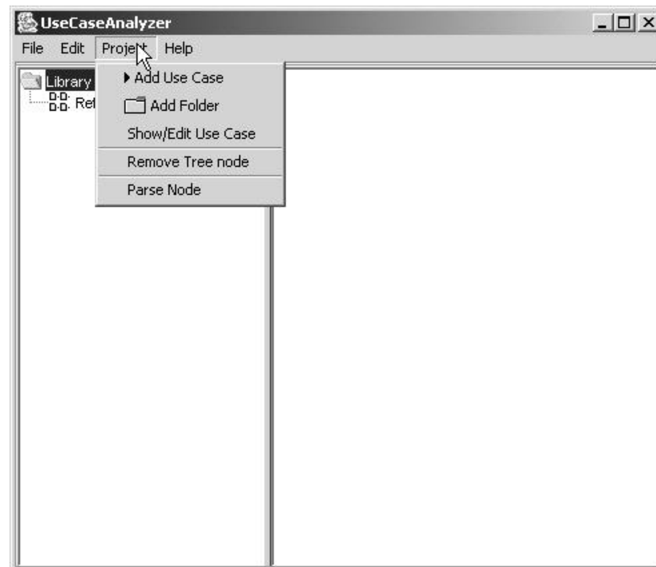

Figure 3.9: UseCaseAnalyzer main GUI

A central part of the UseCaseAnalyzer is the dialog shown in Figure 3.10. This is the link between the GUI and the analyzer engine where each use case is entered into the system. The dialog adds some features like edit, add, or remove sentences belonging to the current use case as well as the possibility to import and export sentences from and to a text file. The text file is named after the title of the use case and each row in the file corresponds to different sentences in the particular use case. When a use case is saved the analyzer of the system scans it to check if there are some grammatical issues the user has to fix before an object model can be made. Saving a use case will actually create one small object model that is only based on the information that can be drawn for that particular use case. Therefore it is possible to check what each and every use case contribute to the combined object model. What actually happens when the combined object model is created is that all included use cases are scanned and all of these object model fragments are merged into the final one. The main purpose for this was to optimize the analyzing work into small chunks that are easy merged together without introduce any duplicate information in the object model. For example if two different use cases extracts vehicle as a class candidate only one class will actually be created (with two traceability references).

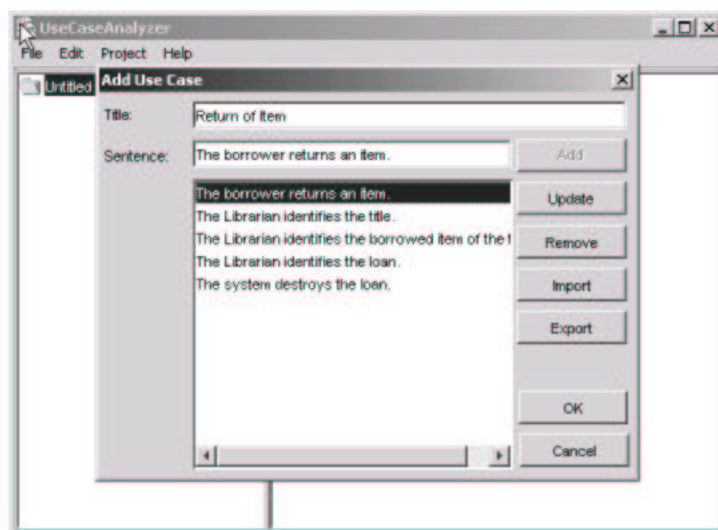

Figure 3.10: Dialog for enter or edit use cases

The UseCaseAnalyzer is implemented with a Java language converter (model transformer) that converts the internal object model to Java programming language. The menu alternative *Parse node* merges all object model fragments (one fragment comes from one use case) together that lies under

the marked node in the tree into one bigger model. In the UseCaseAnalyzer GUI there is a feature to show how this merged object model actually looks like if it is passed through the Java language converter. Figure 3.11 shows an example on how this can look like. The produced Java code is in a way complete and will compile without errors if it is passed through a Java compiler. The Java code produced lacks basically all functionality, since it only contains class frames, stubs for methods and attributes (with get and set methods) and do not contain any code that actually do anything. Functional code can not be extracted from a use case in a reasonable manner and is left for the programmer to write. The purpose with creating code in the first place was never to make the programmer useless but instead gives the programmer a huge support when it comes to creating the base code of a system described in use cases.

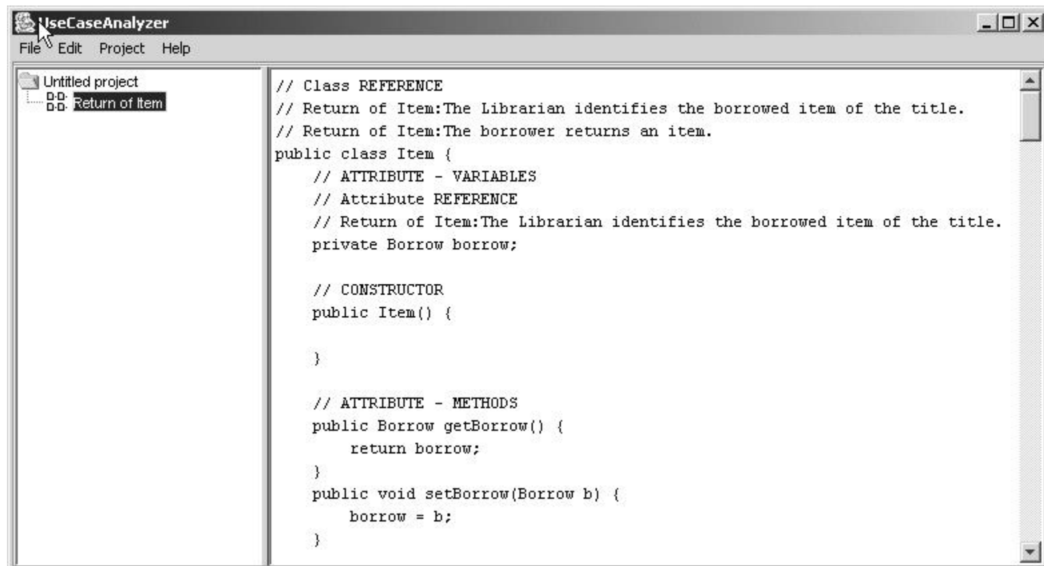

Figure 3.11: Output from a valid use case

3.7 Limitations

As with any first version of a computer program there are limitations in the UseCaseAnalyzer. It should be clear that this is a experimental implementation and, even if its authors have tried to create a good design and a consistent API, it is a bit raw and would be expected to change for it to be truly useful.

The use of LinkGrammar and WordNet introduces external non Java libraries. This breaks platform independence and raises compatibility issues. UseCaseAnalyzer can only run on operating systems supported by both LinkGrammar and WordNet.

Presently LinkGrammar comes with english grammar only and WordNet contain english words only. This effectively constrains the choice of language to use in the use cases to english.

The handling of actors and especially object model entities associated with actors could be better. Actors should not generate an object in the domain model, so the UseCaseAnalyzer does not create objects for the actors specified in the use case. This is an improvement from UseCaseAnalyzers predecessor by Fransson and Larsson, but there is still a problem with this approach. Object model entities, like attributes and methods, can be found and associated with the actor. These potentially important entities will be left out of the object model since the object that they are associated with will never be created. There should be a way of salvaging these entities by associating them with other objects.

The UseCaseAnalyzer does not utilize contextual knowledge. Consider the text “The librarian registers a new book. It has a title, an author, and an ISBN number.”. This text could be used in a library system use case. The words “title”, “author” and, “ISBN number” from the first sentence should generate attributes associated with a ’book’ object derived from the first sentence. Since the UseCaseAnalyzer looks at one sentence at a time and information from previous sentences are not considered it will not create an accurate object model in this case. One solution to this is to restrict the language to disallow this type of discourse in sentences.

Chapter 4

FUJABA

FUJABA is an acronym for "From UML to Java And Back Again". The FUJABA environment aims to provide round trip engineering support for UML and Java ([NNZ00]). The big distinction between other similar tools is that FUJABA has a tight integration with UML class and UML behavior diagrams that enables a visual programming style. The main purpose of this is to enable the user to develop a system on a higher level of abstraction and without any specific knowledge of the Java programming language simply by creating different diagrams and from these diagrams FUJABA will make runnable (compilable) Java code. Many systems can generate code from class diagrams but usually only creates class frames and method declaration without any bodies. FUJABA extends these capabilities by supporting code generation from collaboration diagrams as well (i.e. activity diagrams and state charts). This feature enables the FUJABA system to create code for method bodies and other code blocks.

The USECASEANALYZER extends the FUJABA system with the capability to create use case diagrams that stores the use cases within FUJABA. The USECASEANALYZER extract information from these use cases by using the analyzer part to create Java code that FUJABA can use for creating class diagrams. Figure 4.1 show the integration between parts of the USECASEANALYZER and the FUJABA system where each ellipse corresponds to a use case and a right click (or choosing the menu option *Create Use Case Entry*) brings forth the dialog that enables the user to add, modify or delete properties for that use case.

The FUJABA system always tries to enable traceability information within the diagrams so that if you change some information in one diagram, that change also take place in every other diagram that in some way are related to that information as well. For example if the user changes the name of a class and that class shows up on two different class diagrams, or perhaps an

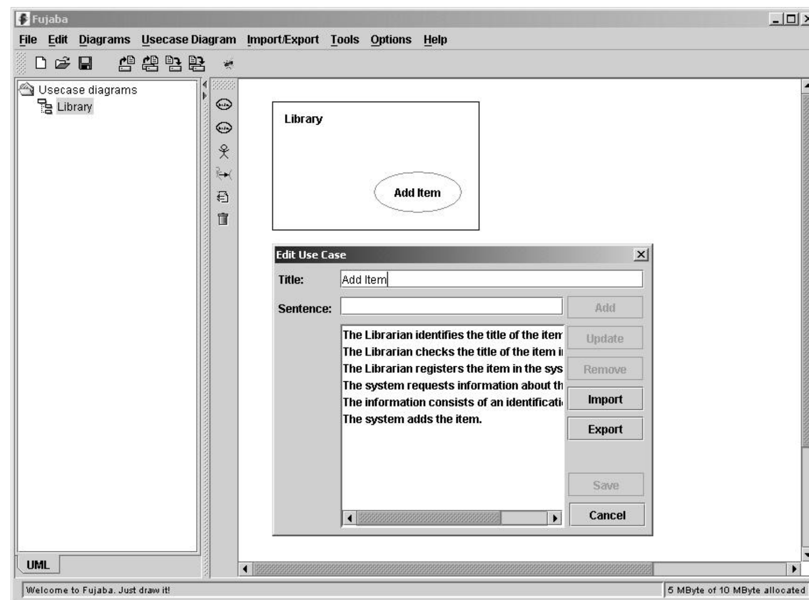

Figure 4.1: The Analyzer integrated in the FUJABA environment

activity diagram, the name will change on all places where that class is used.

When it comes to use cases this is not a simple task to enable since an extracted candidate name does not always corresponds to a single word in a use case. For example the sentence “The title has a title, an author and an ISBN number” from the use case “Add Title” (see [EP98] for the full Library system use case) would create a class candidate ’Title’ with an attribute candidate ’isbnnumber’. These types of problems makes it very difficult to change use cases to reflect changes in some other FUJABA diagrams even if someone wanted to reflect those changes.

The integration of the USECASEANALYZER and the FUJABA system has the approach that all use cases are written before the system design begins, as it is more a part of the requirements gathering rather then the system engineering. This approach makes the tracability references (see Section 3.1) more important because it will be the only way to be able to trace back where the system got each entity name especially if a user of the FUJABA for example changes the entity’s name (or a part of the entity e.g. return types, parameters etc.). The approach to handle this intricate problem is to see use case diagrams a little different from other diagrams in FUJABA. The use case diagrams are a tool to help the system designer with requirement gathering and the other diagrams are to help the user with the design and engineering part. The user should see the connection between use case diagrams and class

diagrams as a one-way non-transitive relationship where added or updated information in a use case diagram will reflect in a totally new class diagram instead of updating an old one. This also means that changes in a class diagram should not be reflected in the use case text that the particular entity is derived from.

Figure 4.2 shows how to create a new class diagram based on a use case diagram. This menu action will, in the case a class diagram already been based on a particular use case diagram, create a new class diagram with the naming *classdiagramname_xxx* where xxx stands for the iteration number.

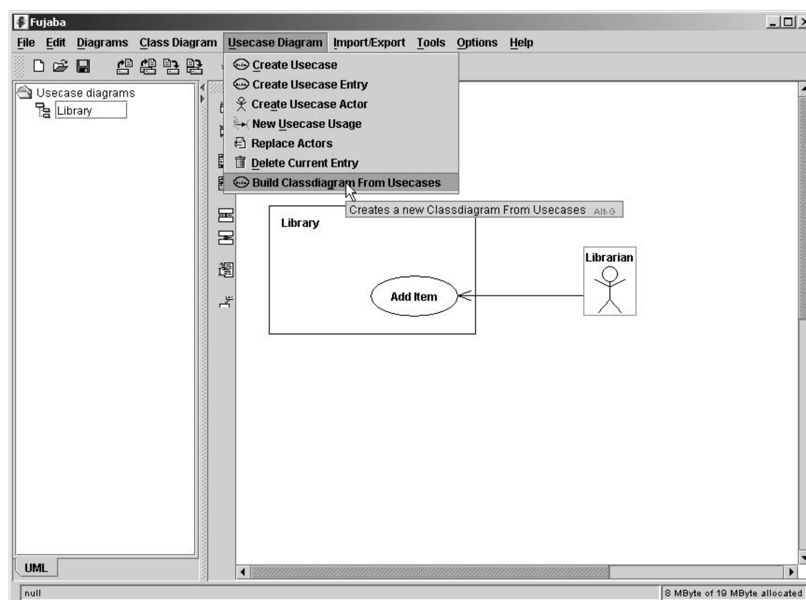

Figure 4.2: How to create a new class diagram from a use case

When a class diagram is created from a use case that class diagram can be used in the same way as ordinary class diagrams within FUJABA where the user can add, modify and removes information as he or she see fit. Figure 4.3 shows an example on how a class diagram that is automatically generated from a use case diagram may look like.

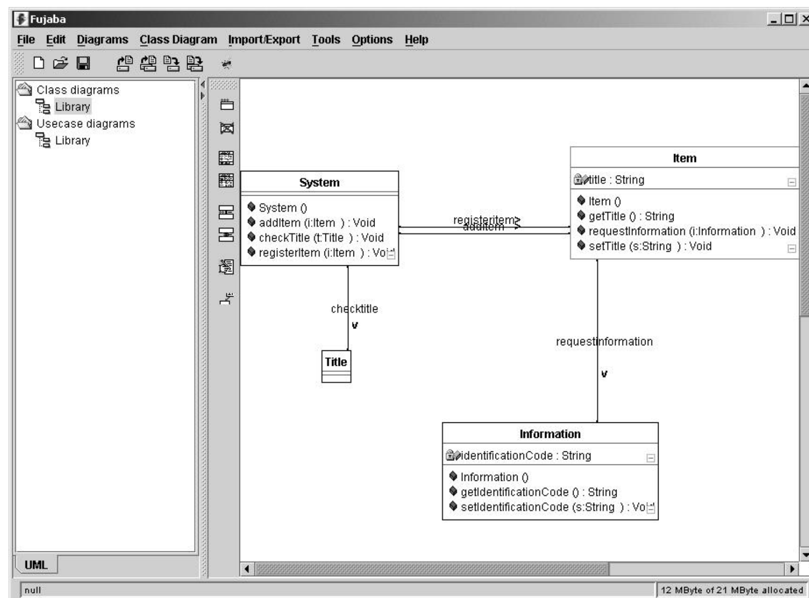

Figure 4.3: The automatically created class diagram from an example use case

Chapter 5

Summary

5.1 Discussion

Use case models has become a popular tool in the software industry since its introduction by Jacobson in 1992. There are several software tools to keep track of use cases and aid with requirement management, but not many of them tries to make the sort of analysis this thesis has described. There might be several reasons for this, but the most likely one is that natural language processing (NLP) is a difficult research area with many complex and partly unresolved issues. While working with this thesis its authors has stumbled on many of these issues and has been forced to acknowledge their own limitations and the limitations imposed by the scope of a master thesis.

The main problem one will face when creating any NLP system is the problem of giving a computer knowledge and understanding. A human understands the meaning of a sentence or paragraph with the aid of her own past experiences and her knowledge about the subject at hand. A sentence can have several semantic meanings and a human is by far better to pick the correct meaning of a sentence than a computer. For example, the sentence "Time flies like an arrow." is seemingly unambiguous, but has actually four different correct interpretations[1]. The reason it seem unambiguous to

---

[1]

- The commonly used meaning; "Time passes as quick as an arrow does"
- a command to time the flies in the same way as an arrow would; "Time flies like an arrow would"
- a command to time only those flies that are like arrows; "Time flies that are like arrows"
- a certain type of flies, namely time-flies, are found of an arrow; "Time-flies appreciate an arrow"

humans are that the three other interpretations makes no sense, but that is something a computer can not determine very easily. Other sentences are impossible to determine even for humans if no further context is given, for example "The woman saw the man on the hill with a telescope."[2]. The subtle ways humans use language to express themselves greatly adds to the challenge of constructing a NLP system. Adding general knowledge to a NLP system is hard but will improve the performance considerably.

Coming up with a rule design that is powerful and simple, and forming suitable rules has been difficult. A rule base with larger expressive power than the regular expression-based approach used by Fransson and Larsson ([FL02]) was desired. Our initial thought was to use first order logic and a theorem prover to represent the rules and make inference over them. Several theorem provers have been evaluated but all of them were turned down, mostly because they where too focused on mathematic theorem proving to be of any use for the purpose of this thesis. Finally the Prolog programming language was chosen for this task. Prolog has the advantage of having implementations in Java so it is easily included in the UseCaseAnalyzer.

The analytic approach to creating rules is not optimal. It would be far better to use a linguistic approach to construct more general rules. Unfortunately none of the authors of this thesis are linguists so the analytic approach was chosen. This was done with the thought that rule creation is a well defined area that could be improved with some external support if time allowed.

5.2 Future work

During the work on this thesis some ideas for improvement on the general approach as well for the UseCaseAnalyzer have come up.

Further usage of WordNet. WordNet has plenty of information about words that are not considered in the current version of the UseCaseAnalyzer. For instance, meronymies (see Section 3.3) could be used to find or confirm attributes of a potential object, or hyponymies could be used to identify parent classes of objects.

Improving the rules. The rules developed for this thesis are biased toward one specific set of use cases. Even if the rules might perform ac-

[2]Who has the telescope, the woman, the man or the hill? Is it the woman or the man who is on the hill?

ceptable on another similar sets they are not very general. The performance of the UseCaseAnalyzer would be greatly improved by creating rules by a more linguistic approach. This might also include enforcing restrictions on the language used in the use cases.

Using controlled language. A common way of dealing with the type of hard problems NLP systems presents is to make the problem easier instead of searching for a perfect solution that might not exist. Restricting the language is one way of making the NLP problem easier at the expense of making the language less natural. If the use cases were described in a controlled language, it would be easier to make better rules since they would not have to be as general. Böttger et. al. ([Böttger+01]) presents a controlled language that in combination with suitable rules could be used to improve the performance of the UseCaseAnalyzer.

Add knowledge. The UseCaseAnalyzer described in this thesis does only have lexical and syntactic knowledge. A NLP system restricted to this type of knowledge is very limited in its parsing power. This is due to the fact that it has no way of making inference based on the meaning of the sentences, it can only identify syntactic structures and try to draw conclusions from them. There are several types of knowledge that could be utilized with varying difficulty (for a brief discussion on this see [FL02] pages 33, 34, 35). It should be known that adding even basic knowledge to a NLP system is far from trivial.

Handling of actors. As mentioned in Section 3.7 the handling of methods and attributes found to belong to actors should be improved. This could be done by finding an algorithm to move the attributes and methods from the actor to a more likely object.

API changes. The API of the UseCaseAnalyzer is in its starting stage and would have to mature some before being considered generally useful.

5.3 Conclusion

This thesis has presented a method for automatic use case analysis. The method is based on grammatical inspection and word knowledge. An experimental tool to test the method has been implemented in Java.

The goals of the thesis was to improve an existing system written by Fransson and Larsson ([FL02]) and to integrate the new and improved system in Fujaba. The implemented tool is called UseCaseAnalyzer, it uses LinkGrammar and WordNet to gather information of the use case sentences and a rule base written in Prolog to make inference over the sentence information. The design of our UseCaseAnalyzer is simpler than the one proposed by Fransson and Larsson and has fewer dependencies to non Java applications.

The object model is stored internally in a meta model. This design makes it easy to export the domain model to Java code, C++ code or any other object oriented language or notation. Currently there is only a Java language converter implemented.

The use of Prolog for representing the rule base is an improvement since it makes the rules more dynamic in nature. The UseCaseAnalyzer is integrated in Fujaba where use cases can be analyzed and the resulting object model shown in a Fujaba class diagram. The first goal of making an improved system was only partly fulfilled, since the UseCaseAnalyzer does not have a more general rule set with better performance than its predecessor. The UseCaseAnalyzer can take actors into accout in its rules, but as the rules are defined in this thesis there are still some problems associated with the handling of actors (see Section 3.7 and 5.2 for details). The second goal of integrating UseCaseAnalyzer into Fujaba is fully met.

We conclude that creating a use case analysis tool that can interpret unrestricted natural language is notoriously difficult, and restrictions on the problem domain is necessary to have stable performance.

Bibliography

[Berard95] E. V. Berard: 'Be Careful With "Use Cases"', The Object Agency, Inc. Gaithersburg, Maryland, 1995

[BKMR90] J. Barnett, K. Knight, I. Mani, E. Rich: 'Knowledge and natural language processing', Communications of the ACM Volume 33 Issue 8, 1990

[Böttger+01] K. Böttger, R. Schwitter, D. Richards, O. Aguilera, D. Molla: 'Reconciling Use Cases via Controlled Languages and Graphical Models', INAP 2001: 186-195

[EP98] H-E. Eriksson, M. Penker: 'UML Toolkit', Wiley computer publishing, 1998

[Firesmith95] D. G. Firesmith: 'Use Cases: The Pros and Cons', Report on Object Analysis and Design (ROAD), 2(2), p. SIGS Publications, New York, New York, p. 2-6, July/August 1995

[FL02] J. Fransson, T. Larsson: 'Use Case Analysis and Model Generation', Department of Computing Science, Umeå University, Sweden, 2002

[Hurlbut97] R. Hurlbut: 'A Survey of Approaches For Describing and Formalizing Use Cases', Illinois Institute of Technology, 1997

[Jacobson92] I. Jacobson: 'Object-Oriented Software Engineering: A Use Case Driven Approach', Addison-Wesley, 1992

[JML00] N. Juristo, A M Moreno, M López: 'How to Use Linguistic Instruments for Object-Oriented Analysis', IEEE Software, May/June 2000

[KG00] D. Kulak, E. Guiney: 'Use cases: Requirements in context', Addison-Wesley, 2000

[LinkGrammar] D. Temperley, D. Sleator, J. Lafferty, 'Link Grammar', (http://www.link.cs.cmu.edu/link/; accessed May 12 2003)

[Miller+] G. A. Miller, R. Beckwith, C. Felbaum, D. Gross, K. Miller: 'Introduction to WordNet: An on-line lexical Database', (http://www.cogsci.princeton.edu/~wn/5papers.ps; accessed May 12 2003)

[NNZ00] U. Nickel, J. Niere, A. Zündorf: 'The FUJABA environment', in Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pp. 241–251, ACM Press, 2000.

[Rosenberg99] D. Rosenberg, 'Use Case Driven Object Modeling with UML', Addison-Wesley, 1999

[Zeien95] G Zeien: 'Weaving a web of use cases', Object Magazine Nov/Dec, 1995

[ÅS] O. Åberg, H. Svensson: 'A text-to-scene converter applied to car accident descriptions', (http://www.efd.lth.se/~e94hsv/html/examensarbete.html; accessed May 12 2003)

Appendix A

The Library use cases

Title: Add Borrower

Actors:Librarian

1. The Librarian requests information about the borrower.
2. This information consists of the name, the address, the zip code, and the state.
3. The borrower information is stored in the system.

Title: Add item

Actors:Librarian

1. The Librarian identifies the title of the item.
2. The Librarian checks the title of the item in the system.
3. The Librarian registers the item in the system.
4. The system requests information about the item.
5. The information consists of an identification bar code.
6. The system adds the item.

Title: Add Title

Actors:Librarian

1. The Librarian adds a title to the system.
2. The title has a title, an author and an ISBN number.

Title: Lend item

Actors:Librarian

1. The Librarian identifies the title and an available item.
2. The Librarian identifies the borrower.
3. The library lends the item to the borrower.
4. The Librarian registers a new loan in the system.
5. The Librarian removes any reservation of the title.

Title: Make reservation

Actors:Librarian

1. The Librarian identifies the title.
2. The Librarian identifies the borrower.
3. The Librarian creates a reservation containing the title and borrower.

Title: Remove item

Actors:Librarian

1. The Librarian identifies the title of the item.
2. The system requests the identification bar code of the item.
3. The system removes the item, and loans of the item.

Title: Remove Reservation

Actors:Librarian

1. The Librarian identifies the borrower.
2. The Librarian identifies the reserved title.
3. The Librarian removes the reservation.

Title: Remove or Update Borrower

Actors:Librarian

1. The system requests the name of the borrower.
2. The system requests loans or reservations of the borrower.
3. The Librarian removes the borrower information from the system.
4. The Librarian removes loans or reservations of the borrower.
5. The Librarian specifies the borrower.
6. The system displays information about the borrower.
7. The Librarian changes the information about the borrower.
8. The system updates the information.

Title: Remove or Update Title

Actors:Librarian

1. The Librarian specifies the title.
2. The system removes all items of the title.
3. The system removes all loans of the items.
4. The system removes the title.
5. The Librarian specifies the title.
6. The system displays information about the title.
7. The Librarian changes the information about the title.
8. The system updates the information.

Title: Return of item

Actors:Librarian

1. The borrower returns an item.
2. The Librarian identifies the title.
3. The Librarian identifies the borrowed item of the title.
4. The Librarian identifies the loan.
5. The system destroys the loan.